Criptografía

Práctica 2

Pareja 07:

José Luis Suárez Fueyo

Javier Santos Lorenzo

Introducción

El objetivo de esta práctica es el entendimiento de los conceptos de entropía, seguridad perfecta y con los métodos de cifrado simétrico como Data Encryption Standard (DES) y Advanced Encryption Standard (AES).

En la memoria se ha seguido el siguiente esquema:

- Marco teórico: con el fin de relacionar los conocimientos adquiridos en las clases de teoría con el código a desarrollar en la práctica, se decide realizar una breve explicación teórica en cada uno de los ejercicios

- Código: en cada apartado se incluyen las funciones principales relativas de ese apartado. El resto del código (principalmente funciones auxiliares) se omite para evitar alargar la memoria innecesariamente.

- Resultados: en cada apartado se explican las diferentes pruebas realizadas, las conclusiones obtenidas, así como se da respuesta a las posibles preguntas planteadas en el enunciado de la práctica.

En la práctica se incluyen los ficheros de código, la memoria y diversos ficheros de entrada que se han realizado para las pruebas, así como los ficheros de salida generados.

Ejercicio 1: Seguridad perfecta

Marco teórico

En este apartado se pretende verificar si el cifrado afín consigue Seguridad Perfecta en caso de utilizar claves equiprobables y no se consiguen cuando las claves se distribuyen de manera aleatoria.

Recordamos que un criptosistema tiene Seguridad Perfecta si se verifica:

$$P_p(x) = P_p(x|y) \quad \forall x \in X \ \forall y \in Y$$

Código

En este apartado hay tres funciones destacadas:

La primera de ellas es la que inicializa las probabilidades iguales (caso de claves equiprobables).

```
void iniProbsIguales(struct TABLA_PROBS* p, int m) {
  int i;
  p->m = m;
  p->probabilidades = malloc(sizeof(double) * m);
  for (i = 0; i < m; i++) p->probabilidades[i] = 1.0 / m;
  p->condicionadas = NULL;
```

La segunda es la que inicializa las probabilidades de manera aleatoria (caso de claves aleatorias).

```
void iniProbsRandom(struct TABLA_PROBS* p, int m) {
 int i;
 double total = 0;
 p->m = m;
 p->probabilidades = malloc(sizeof(double) * m);
 for (i=0; i < m; ++i) p->probabilidades[i] = (rand()+1.0) / RAND_MAX;
 for (i=0; i < m; ++i) total += p->probabilidades[i];
 for (i=0; i < m; ++i)  p->probabilidades[i] /= total;
 p->condicionadas = NULL;
}
```

Por último, siendo la función más destacada del apartado, está la función que se utiliza para obtener las probabilidades.

```
int obtenerprobabilidades(struct TABLA_PROBS* resultado, FILE* fentrada, struct TABLA_PROBS* probs, int repeticiones) {
  int i,j;
  int m = probs->m;//Tam alfabeto
  double aux;
  double *frecuenciaCifrado = NULL;
  int clave, charplano, charcifrado;
  char c;
  struct TABLA_PROBS pCifrado;
  //Preparamos la estructura de texto cifrado
  frecuenciaCifrado = malloc(sizeof(double) * m);
  pCifrado.probabilidades = malloc(sizeof(double) * m);
  pCifrado.m = m;
  //Limpiamos las estructuras por si hay escritura previa en la memoria y calculas las probabilidades de cifrado
  for (i = 0; i < m; ++i) frecuenciaCifrado[i] = 0;
  pCifrado.probabilidades[0] = probs->probabilidades[0];
  for (i = 1; i < m; ++i) pCifrado.probabilidades[i] = pCifrado.probabilidades[i-1] + probs->probabilidades[i];
  for (i = 0; i < m; ++i) resultado->probabilidades[i] = 0;
  for (i = 0; i < m; ++i) for (j=0; j < m; ++j) resultado->condicionadas[i][j] = 0;
  aux = 0;
  for (i = 0; i < repeticiones; ++i) {
        //Preparamos el puntero a principio del fichero
        rewind(fentrada);
        //Calculamos la frecuencia y las probabilidades de cifrado
        while (((c = fgetc(fentrada)) != EOF) && (fentrada != stdin || c != '\n')) {
              clave = claveAleatoria(&pCifrado);
              charplano = getCharPlano(c, m);
              if (charplano < 0 || charplano >= m) continue;
```

```
                else aux++;
                charcifrado = (charplano + clave) % m;
                resultado->probabilidades[charplano]++;
                resultado->condicionadas[charplano][charcifrado]++;
                frecuenciaCifrado[charcifrado]++;
        }
  }
  //Divide entre los caracteres totales, para obtener la probabilidad (casos favorables/casos totales)
  for (i=0; i < m; ++i) resultado->probabilidades[i] /= aux;
  for (i=0; i < m; ++i) for (j=0; j < m; ++j) resultado->condicionadas[i][j] = frecuenciaCifrado[j]? resultado-
>condicionadas[i][j]/frecuenciaCifrado[j]:0;
  //Liberamos la memoria que hemos usado en la funcion
  free(frecuenciaCifrado);
  free(pCifrado.probabilidades);
  return 0;
}
```

Resultados

Para probar este apartado se utiliza la siguiente instrucción:

./1 -P|-I -i entrada.txt -o salida.txt, donde

- -P: indica que se utilizan claves equiprobables
- -I: modo de cifrado no equiprobable.
- -i entrada.txt : Nombre del fichero de entrada. Se aconseja utilizar un fichero lo más largo posible, para obtener probabilidades más fiables.
- -o salida.txt: Nombre del fichero de salida. Pese a que se deja a voluntad de la persona que ejecuta el código si incluir un fichero de salida o realizar la impresión en la terminal, se recomienda crear un fichero de salida para poder verificar más cómodamente si se cumple el principio de Seguridad Perfecta. En nuestro caso, se han creado los ficheros SegPerf.txt (cuando usamos P como modo de cifrado) y NoSegPerf.txt (cuando usamos I).

El resto de ejercicios seguirán una metodología bastante similar a este, por lo que solo se explica en este apartado el significado de cada parte de la instrucción.

El programa comienza contando las repeticiones de cada una de las letras en el texto plano, para calcular la probabilidad de ocurrencia de cada una de ellas dividiendo el número de apariciones entre la longitud total del texto.

Después se cifra el texto plano. El programa generará un k número aleatorio para cada carácter y cifrará, mediante el cifrado César, ese carácter con la clave k.

Una vez que se ha cifrado el texto, se calculan las probabilidades condicionadas utilizando la fórmula de Bayes vista en teoría.

Finalmente se imprimen las probabilidades en un fichero de salida (o por la terminal). Cuando se utiliza el modo P (claves equiprobables), se genera el fichero SegPerf.txt en el cual se verifica que tiene Seguridad Perfecta. Para el caso de las claves aleatorias (modo I), se genera el fichero NoSegPerf.txt en el cual se puede comprobar fácilmente que no se verifica el principio de Seguridad Perfecta.

Se han probado diferentes textos para este ejercicio, dando mejores resultados el último utilizado (el Quijote), dado su gran extensión. Para textos cortos (como copiando el primer gran párrafo del Quijote), y utilizando el modo P, no se verifica la Seguridad Perfecta. Esto es debido al escaso tamaño del texto, haciendo que $P_p(x)$ sea distinto de $P_{\underline{p}}(x|y)$.

Ejercicio 2: Implementación del DES

a. Programación del DES

Marco teórico

En esta apartado se nos pide la implementación de DES de 16 rondas tanto en el modo CBC, como en el modo ECB, este último con el fin de comprobar que no es un método bueno para mensajes largos y estructurados, como podría ser una imagen.

A continuación, se va a realizar una pequeña explicación de ambos modos:

- ECB: cada bloque de 64 bits se cifra de manera independiente usando la misma clave. Por eso, mismos textos de entrada serán cifrados con mismos textos de salida. Por esta razón, no es un método bueno para mensajes largos y resultados, como comprobaremos en la sección de resultados.

- CBC: a cada bloque de texto plano se le aplica la operación xor con el bloque de cifrado anterior antes de ser cifrado. De esta manera, cada bloque de texto cifrado depende del texto plano hasta entonces procesado. Asimismo, se utiliza un vector de inicialización (IV), permitiendo lograr un mensaje único. IV es secreto y formará parte de la clave.

  Para el cifrado y descifrado se tienen las siguientes recursiones:

  $C_1 = E_k(IV \oplus P_1)$
  $C_n = E_k(C_{n-1} \oplus P_n)$ con n mayor que 1
  $P_1 = IV \oplus D_k(C_1)$
  $C_n = C_{n-1} \oplus E_k(C_n)$ con n mayor que 1

Código

Cabe destacar que el código utilizado para ambos modos es muy similar, por ello solo se incluirán las funciones del CBC, la mayoría de las cuales son iguales a ECB, teniendo otras alguna funcionalidad extra.

La primera realiza el PC1 del DES, permutando y comprimiendo la clave:

```
void aplicarPC1(uint8_t* clave, DescomposicionClave* subclaves) {
  int i, desp;
  uint8_t byte_desp;

  for (i = 0; i < BITS_IN_PC1; i++) {
        desp = PC1[i];
        //Mascara para tomar el bit de la posicion correcta de la clave
        byte_desp = 0x80 >> ((desp - 1) % 8);
        //Cogemos el bit de nuestra clave
        byte_desp &= clave[(desp - 1) / 8];
        //Si era un 1 lo movemos al principio si era 0 se queda asi
        byte_desp <<= ((desp - 1) % 8);
        //Metemos en nuestra subclave el bit obtenido en la posicion que le corresponde (mod 8)
        subclaves[0].k[i / 8] |= (byte_desp >> i % 8);
  }

}
```

La segunda realiza la división de la clave K en las mitades C y D del esquema de la teoría.

```
void dividirClave(DescomposicionClave* subclaves) {
  int i;
  //C corresponde a los 28 primeros bits de K
```

```
  for (i = 0; i < 3; i++) {
        subclaves[0].c[i] = subclaves[0].k[i];
  }
  //Por tanto llega hasta la primera mitad del cuarto byte de K
  subclaves[0].c[3] = subclaves[0].k[3] & 0xF0;

  //D usa los 28 bits siguientes
  for (i = 0; i < 3; i++) {
        subclaves[0].d[i] = (subclaves[0].k[i + 3] & 0x0F) << 4;
        subclaves[0].d[i] |= (subclaves[0].k[i + 4] & 0xF0) >> 4;
  }
  subclaves[0].d[3] = (subclaves[0].k[6] & 0x0F) << 4;
}
```

La siguiente realiza el desplazamiento circular a la izquierda de las subclaves.

```
void aplicarLCS(DescomposicionClave* subclaves) {
  int i, j, desp;
  uint8_t byte_desp, bits_desp1, bits_desp2, bits_desp3, bits_desp4;

  //En cada una de las rondas
  for (i = 1; i < ROUNDS + 1; i++) {
        //Rellenamos con la subclave anterior
        for (j = 0; j < 4; j++) {
              subclaves[i].c[j] = subclaves[i - 1].c[j];
              subclaves[i].d[j] = subclaves[i - 1].d[j];
        }
        //Obtenemos el desplazamiento de la ronda
        desp = ROUND_SHIFTS[i];
        if (desp == 1) {
              byte_desp = 0x80;
        } else {
              byte_desp = 0xC0;
        }

        // Aqui tomo los bits que voy a desplazar en C
        bits_desp1 = byte_desp & subclaves[i].c[0];
        bits_desp2 = byte_desp & subclaves[i].c[1];
        bits_desp3 = byte_desp & subclaves[i].c[2];
        bits_desp4 = byte_desp & subclaves[i].c[3];

        //Desplazo el primero lo necesario a la izquierda y en sus huecos meto los desplazados del anterior
        subclaves[i].c[0] <<= desp;
        subclaves[i].c[0] |= (bits_desp2 >> (8 - desp));
        //Para el segundo
        subclaves[i].c[1] <<= desp;
        subclaves[i].c[1] |= (bits_desp3 >> (8 - desp));
        //Para el tercero
        subclaves[i].c[2] <<= desp;
        subclaves[i].c[2] |= (bits_desp4 >> (8 - desp));
        //Para el cuarto hay que tener en cuenta que es circular y que ademas solo valen los 4 primeros bits
        subclaves[i].c[3] <<= desp;
        subclaves[i].c[3] |= (bits_desp1 >> (4 - desp));

        // Igual para D
        bits_desp1 = byte_desp & subclaves[i].d[0];
        bits_desp2 = byte_desp & subclaves[i].d[1];
        bits_desp3 = byte_desp & subclaves[i].d[2];
        bits_desp4 = byte_desp & subclaves[i].d[3];

        subclaves[i].d[0] <<= desp;
        subclaves[i].d[0] |= (bits_desp2 >> (8 - desp));

        subclaves[i].d[1] <<= desp;
        subclaves[i].d[1] |= (bits_desp3 >> (8 - desp));

        subclaves[i].d[2] <<= desp;
        subclaves[i].d[2] |= (bits_desp4 >> (8 - desp));

        subclaves[i].d[3] <<= desp;
        subclaves[i].d[3] |= (bits_desp1 >> (4 - desp));
```

```
        for (j = 0; j < BITS_IN_PC2; j++) {
                //Lo mismo para PC2 con 48 bits para las subclaves
                desp = PC2[j];
                 //Si es menor o igual que 28 es que estoy en C
                if (desp <= BITS_IN_PC1 / 2) {
                        byte_desp = 0x80 >> ((desp - 1) % 8);
                        byte_desp &= subclaves[i].c[(desp - 1) / 8];
                        byte_desp <<= ((desp - 1) % 8);
                } else { //Si no estoy en D
                        byte_desp = 0x80 >> ((desp - 29) % 8);
                        byte_desp &= subclaves[i].d[(desp - 29) / 8];
                        byte_desp <<= ((desp - 29) % 8);
                }
                subclaves[i].k[j / 8] |= (byte_desp >> j % 8);
        }

  }
}
```

La siguiente aplica el DES, realizando la funcionalidad de la parte izquierda del esquema de la teoría.

```
void aplicarDES(uint8_t* bloque_entrada, uint8_t* bloque_salida,
        DescomposicionClave* subclaves, int modo, uint8_t * chain) {
  uint8_t aux_ip[8], l[4], r[4], IV[8];
  memset(aux_ip, 0, 8);
  memset(bloque_salida, 0, 8);

  newClave(IV);
  //En este caso es cuando se cifra
  if (modo == 1)
        xorIV(bloque_entrada, chain);
  //Realizamos la permutacion inicial
  aplicarIP(bloque_entrada, aux_ip);
  //Realizamos las rondas del DES
  aplicarRondaDES(aux_ip, modo, subclaves, r, l);
  //Realizamos la inversa de la permutacion inicial
  aplicarInversaIP(bloque_salida, aux_ip, r, l);
  //Este es el caso de cuando se descifra
  if (modo == 2)
        xorIV(bloque_salida, chain);

  return;

}
```

Esta función realiza la permutación inicial del DES

```
void aplicarIP(uint8_t* bloque_entrada, uint8_t* aux_ip) {

  int desp, i;
  uint8_t byte_desp;

  //Seguimos un  procedimiento similar a con las claves
  for (i = 0; i < BITS_IN_IP; i++) {
        desp = IP[i];
        byte_desp = 0x80 >> ((desp - 1) % 8);
        byte_desp &= bloque_entrada[(desp - 1) / 8];
        byte_desp <<= ((desp - 1) % 8);

        aux_ip[i / 8] |= (byte_desp >> i % 8);
  }

  return;

}
```

Esta función realiza la inversa de la permutación inicial del DES

```
void aplicarInversaIP(uint8_t* bloque_salida, uint8_t* aux_ip, uint8_t* r,
        uint8_t* l) {

  int desp, i;
  uint8_t byte_desp;

  //Concatenamos invirtiendo el orden antes de aplicar IP_INV
  for (i = 0; i < 4; i++) {
        aux_ip[i] = r[i];
        aux_ip[4 + i] = l[i];
  }

  for (i = 0; i < BITS_IN_IP; i++) {
        desp = IP_INV[i];
        byte_desp = 0x80 >> ((desp - 1) % 8);
        byte_desp &= aux_ip[(desp - 1) / 8];
        byte_desp <<= ((desp - 1) % 8);
        bloque_salida[i / 8] |= (byte_desp >> i % 8);
  }

  return;

}
```

Por último, esta función realiza la funcionalidad de cada ronda del DES

```
void aplicarRondaDES(uint8_t* aux_ip, int modo, DescomposicionClave* subclaves,
        uint8_t* r, uint8_t* l) {

  int i, k, desp, orden_clave;
  uint8_t li[4], ri[4], er[6], sbox_er[4], fila, col, byte_desp;

  //Una vez permutada la clave se divide en parte izquierda y parte derecha
  for (i = 0; i < 4; i++) {
        l[i] = aux_ip[i];
        r[i] = aux_ip[i + 4];
  }

  for (k = 1; k <= ROUNDS; k++) {
        //Li=Ri-1
        memcpy(li, r, 4);
        memset(er, 0, 6);

        //Con E procedemos como con IP
        for (i = 0; i < BITS_IN_E; i++) {
              desp = E[i];
              byte_desp = 0x80 >> ((desp - 1) % 8);
              byte_desp &= r[(desp - 1) / 8];
              byte_desp <<= ((desp - 1) % 8);
              er[i / 8] |= (byte_desp >> i % 8);
        }

        if (modo == 2) {
              orden_clave = 17 - k;
        } else {
              orden_clave = k;
        }

        //F=E(Ri-1) XOR Ki
        for (i = 0; i < 6; i++) {
              er[i] ^= subclaves[orden_clave].k[i];
        }

        memset(sbox_er, 0, 4);

              //Se van rellenando los 24 bits de la salida de las sboxes teniendo en cuenta por el que nos ibamos para
aplicar correctamente las mascaras

        // Byte 1
        fila = 0;
```

```
fila |= ((er[0] & 0x80) >> 6);
fila |= ((er[0] & 0x04) >> 2);

col = 0;


col |= ((er[0] & 0x78) >> 3);

sbox_er[0] |= ((uint8_t) S_BOXES[0][fila][col] << 4);

fila = 0;
fila |= (er[0] & 0x02);
fila |= ((er[1] & 0x10) >> 4);

col = 0;
col |= ((er[0] & 0x01) << 3);
col |= ((er[1] & 0xE0) >> 5);

sbox_er[0] |= (uint8_t) S_BOXES[1][fila][col];

// Byte 2
fila = 0;
fila |= ((er[1] & 0x08) >> 2);
fila |= ((er[2] & 0x40) >> 6);

col = 0;
col |= ((er[1] & 0x07) << 1);
col |= ((er[2] & 0x80) >> 7);

sbox_er[1] |= ((uint8_t) S_BOXES[2][fila][col] << 4);

fila = 0;
fila |= ((er[2] & 0x20) >> 4);
fila |= (er[2] & 0x01);

col = 0;
col |= ((er[2] & 0x1E) >> 1);

sbox_er[1] |= (uint8_t) S_BOXES[3][fila][col];

// Byte 3
fila = 0;
fila |= ((er[3] & 0x80) >> 6);
fila |= ((er[3] & 0x04) >> 2);

col = 0;
col |= ((er[3] & 0x78) >> 3);

sbox_er[2] |= ((uint8_t) S_BOXES[4][fila][col] << 4);

fila = 0;
fila |= (er[3] & 0x02);
fila |= ((er[4] & 0x10) >> 4);

col = 0;
col |= ((er[3] & 0x01) << 3);
col |= ((er[4] & 0xE0) >> 5);

sbox_er[2] |= (uint8_t) S_BOXES[5][fila][col];

// Byte 4
fila = 0;
fila |= ((er[4] & 0x08) >> 2);
fila |= ((er[5] & 0x40) >> 6);

col = 0;
col |= ((er[4] & 0x07) << 1);
col |= ((er[5] & 0x80) >> 7);

sbox_er[3] |= ((uint8_t) S_BOXES[6][fila][col] << 4);

fila = 0;
```

```
        fila |= ((er[5] & 0x20) >> 4);
        fila |= (er[5] & 0x01);

        col = 0;
        col |= ((er[5] & 0x1E) >> 1);

        sbox_er[3] |= (uint8_t) S_BOXES[7][fila][col];

        memset(ri, 0, 4);

        for (i = 0; i < BITS_IN_P; i++) {
                desp = P[i];
                byte_desp = 0x80 >> ((desp - 1) % 8);
                byte_desp &= sbox_er[(desp - 1) / 8];
                byte_desp <<= ((desp - 1) % 8);

                ri[i / 8] |= (byte_desp >> i % 8);
        }

        for (i = 0; i < 4; i++) {
                ri[i] ^= l[i];
        }

        for (i = 0; i < 4; i++) {
                l[i] = li[i];
                r[i] = ri[i];
        }
  }
}
```

Resultados

La manera de ejecutar ambos modos es la siguiente:

- ECB: Uso: ./desECB {-C | -D -k clave} [-i fichero entrada] [-o fichero salida].
- CBC: Uso: ./desCBC {-C | -D -k clave -IV vector inicializacion } [-i fichero entrada] [-o fichero salida].

Para ambos se han realizado pruebas similares. Partiendo tanto de un texto largo, como el Quijote (entrada.txt) o de una imagen (asturias.jpg), se ha cifrado el fichero, procedimiento posteriormente al descifrado del fichero cifrado. Al realizar este proceso, se ha obtenido el fichero original, como puede verse en los ficheros entregados (SaldesC1.txt caso del texto y fotoCBC.jpg caso de la imagen). Con ello, se puede concluir que ambos modos son correctos, pues cifran y descifran correctamente.

Por último, en el caso de ECB, además de comprobar que cifraba y descifraba correctamente, se intentó mostrar que no borra la estructura de la imagen, como la imagen proporcionada en el enunciado. Sin embargo, cuando se cifraba el fichero que se generaba aparecía como corrompido, aunque luego si se aplicara el descifrado se obtuviera el original. No se sabe, si esto es debido a la extensión que utilizábamos o por qué ocurría esto. Por este motivo, no se ha podido incluir una imagen similar a la del enunciado.

Ejercicio 3: Implementación del DES

a. Estudio de la no linealidad de las S-boxes del DES

Marco teórico

En el DES se verifica el criterio de no linealidad de las S-boxes, esto es, S(x ⊕ y) = S(x)⊕S(y). Se impone este criterio con el fin de complicar el criptoanálisis del método. A lo largo de este apartado se ha comprobado que se cumple la no linealidad.

Código

A continuación, se añaden las principales funciones de este apartado:

La primera de ellas, es la que mide la no linealidad de las S-boxes del DES, contando el número de bits iguales entre S(x ⊕ y) = S(x)⊕S(y)..

```
void getDatos(unsigned short*** Sboxes, unsigned long int* coincidencias) {

  BloqueDES x, y, S_xXORy, S_xXORS_y;
  BloqueDES xXORy, S_x, outputY;
  int i, j;
  int coincidences;

  //Preparamos un par de bloques nuevos par aprobar la linealidad
  //de entrada para las cajas
  nuevoBloque(&x, SBOX_INPUT_BITS * NUM_S_BOXES);
  nuevoBloque(&y, SBOX_INPUT_BITS * NUM_S_BOXES);

  //Aplicamos la suma en la caja S(x+y)
  xorDES(&xXORy, &x, &y, SBOX_INPUT_BITS * NUM_S_BOXES);  //x+y
  SBoxGeneral(&S_xXORy, &xXORy, Sboxes);   //S(x+y)

  //Aplicamos la suma fuera de la caja  S(x)+S(y)
  SBoxGeneral(&S_x, &x, Sboxes);         //S(x)
  SBoxGeneral(&outputY, &y, Sboxes);   //S(y)
  xorDES(&S_xXORS_y, &S_x, &outputY, SBOX_INPUT_BITS * NUM_S_BOXES);//S(x)+S(y)

  for (i = 0; i < NUM_S_BOXES; i++) {
        coincidences = 0;
        for (j = 0; j < SBOX_OUTPUT_BITS; j++) {
              if (S_xXORy.bloque[i * SBOX_OUTPUT_BITS + j + 1]
                          != S_xXORS_y.bloque[i * SBOX_OUTPUT_BITS + j + 1])
                    coincidences++;
        }
        (coincidencias[(SBOX_OUTPUT_BITS + 1) * i + coincidences])++;
  }
}
```

La segunda función destacada es la que calcula la esperanza, la desviación típica y las coincidencias, fundamental para el análisis de los resultados.

```
void calcularEstadisticas(NL* resultados, int nPruebas) {

  unsigned short*** Sboxes = NULL;
  unsigned long int coincidencias[NUM_S_BOXES * (SBOX_OUTPUT_BITS + 1)];
  int i, j;
  float aux[SBOX_OUTPUT_BITS + 1];
  float temp;

  srand(time(NULL));

  //Prepramos la estructura donde anotaremos las coincidencias
```

```
  for (i = 0; i < NUM_S_BOXES * (SBOX_OUTPUT_BITS + 1); i++)
        coincidencias[i] = 0;

  //Calculamos las distribuciones de linealidad
  Sboxes = guardarMemSboxes();//Preparamos las cajas
  for (i = 0; i < nPruebas; i++)
        getDatos(Sboxes, coincidencias); //Calculamos coincidencias
  freeSboxes(Sboxes);

  //Procesa los resultados
  for (j = 0; j < SBOX_OUTPUT_BITS + 1; j++)
        aux[j] = j * (1. / SBOX_OUTPUT_BITS);

  for (i = 0; i < NUM_S_BOXES; i++) {
        //Calculamos el porcentaje de coincidencias
        for (j = 0; j < SBOX_OUTPUT_BITS + 1; j++)
              resultados->coincidencias[(SBOX_OUTPUT_BITS + 1) * i + j] =
                    coincidencias[(SBOX_OUTPUT_BITS + 1) * i + j]
                          * (1. / nPruebas);

        //Calcula la esperanza
        resultados->expectation[i] = 0;
        for (j = 0; j < SBOX_OUTPUT_BITS + 1; j++)
              resultados->expectation[i] +=
                    (aux[j]
                          * resultados->coincidencias[(SBOX_OUTPUT_BITS + 1)
                                * i + j]);

        //Calculamos la desviacion tipica
        resultados->desviacion[i] = 0;
        for (j = 0; j < SBOX_OUTPUT_BITS + 1; j++) {
              temp = (aux[j] - resultados->expectation[i]);
              resultados->desviacion[i] += temp * temp
                    * resultados->coincidencias[(SBOX_OUTPUT_BITS + 1) * i + j];
        }
        resultados->desviacion[i] = sqrt(resultados->desviacion[i]); //Nos falta aplicar la raiz cuadrada
  }
}
```

Resultados

Se ha generado el fichero Nldes.txt (mediante la siguiente instrucción: ./NL_DES {-n numero de pruebas} [-o fichero salida]) con los resultados obtenidos, los cuales pasamos a analizar:

S-box	Esperanza	Desviación típica
1	0.53	0.23
2	0.51	0.24
3	0.47	0.27
4	0.56	0.22
5	0.53	0.26
6	0.54	0.26
7	0.52	0.26

8	0.52	0.25

En el cuadro anterior se puede ver como la linealidad es muy cercana a 1⁄2 en todas las S-boxes, con una desviación típica pequeña (1⁄4). Esto nos indica que, en promedio, las cajas están a medio camino entre ser completamente lineales y completamente no lineales, no aportando ninguna información de este modo para el criptoanálisis.

Como la desviación típica es pequeña, los valores son próximos a la media. Por ello, en la próxima tabla, mostramos los bits de diferencia entre S(x⊕y) y S(x) ⊕ S(y).

S-box	**0 cambios**	**1 cambios**	**2 cambios**	**3 cambios**	**4 cambios**
1	2%	25%	37%	31%	5%
2	6%	25%	38%	20%	11%
3	9%	23%	45%	19%	4%
4	3%	21%	26%	47%	3%
5	5%	25%	29%	34%	7%
6	6%	20%	33%	32%	9%
7	3%	32%	26%	31%	8%
8	3%	26%	33%	29%	9%

Como puede verse observando las tablas, en general, las S-boxes tienen una distribución bastante gausiana del número de bits de diferencia.

b. Estudio del Efecto Avalancha

Marco teórico

El Efecto Avalancha se basa en la idea de que un cambio de bit en el bloque de texto plano o de la clave produce un gran número de cambios en los bits del texto cifrado. Esto es debido a que los cambios se propagan a lo largo de las sucesivas rondas del DES, de manera que tras cada ronda el número de bits diferentes aumenta hasta estabilizarse en torno a un valor óptimo.

Por ello, en este ejercicio se probará a realizar diversos cambios en la clave y el texto plano y se comprobará cómo afectan dichos cambios al cifrado.

Código

Se muestran las funciones más destacadas. La primera calcula los bits diferentes al cambiar un texto, realizando el proceso de DES necesario. Cabe destacar que tanto esta función como la

siguiente llaman a una serie de funciones que realizan la lógica del DES. Dichas funciones se limitan a utilizar las variables proporcionadas en Moodle. Es por ello que no se incluye aquí en la memoria.

```
void pruebaTextoCambiado(BloqueDES* entrada, BloqueDES* clave,
        unsigned long int* bitsCambiado, short unsigned int*** Sboxes) {

  BloqueDES bloqueIzq_pre, bloqueDer_pre, bloqueIzq_post, bloqueDer_post;
  BloqueDES bloqueTrasIP;
  BloqueDES clave_pre, clave_post, clave_ronda;
  BloqueDES entrada2;
  BloqueDES bloqueIzq2_pre, bloqueDer2_pre, bloqueIzq2_post, bloqueDer2_post;
  BloqueDES bloqueTrasIP2;
  int i;

  //Vamos a preparar la entrada cambiada
  copiarBloque(&entrada2, entrada, TAM_BLOQUE);
  cambioBit(&entrada2, TAM_BLOQUE);
  bitsCambiado[0] += calculaDiferencias(entrada, &entrada2, TAM_BLOQUE);

  //Aplicamos la permutacion inicial a la entrada y la entrada cambiada
  permInicial(&bloqueTrasIP, entrada);
  permInicial(&bloqueTrasIP2, &entrada2);

  //Dividimos los dos bloques ne mitades
  getParteIzquierda(&bloqueIzq_pre, &bloqueTrasIP);
  getParteDerecha(&bloqueDer_pre, &bloqueTrasIP);
  getParteIzquierda(&bloqueIzq2_pre, &bloqueTrasIP2);
  getParteDerecha(&bloqueDer2_pre, &bloqueTrasIP2);

  //PC1
  aplicarPC1(&clave_pre, clave);

  //Aplicamos las rondas DES por duplicado
  for (i = 1; i <= NUM_ROUNDS; i++) {
        //Aplicamos LCS
        aplicarLCS(&clave_post, &clave_pre, i, 1);
        //Aplicamos PC2
        aplicarPC2(&clave_ronda, &clave_post);

        //Aplicamos DES por duplicado
        aplicarDES(&bloqueIzq_post, &bloqueDer_post, &bloqueIzq_pre, &bloqueDer_pre, &clave_ronda,
                i, Sboxes);
        aplicarDES(&bloqueIzq2_post, &bloqueDer2_post, &bloqueIzq2_pre, &bloqueDer2_pre,
                &clave_ronda, i, Sboxes);

        //Calculamos diffs
        bitsCambiado[i] += calculaDiferencias(&bloqueIzq_post, &bloqueIzq2_post,
        BITS_IN_FEISTEL / 2);
        bitsCambiado[i] += calculaDiferencias(&bloqueDer_post, &bloqueDer2_post,
        BITS_IN_FEISTEL / 2);

        //Prepramos los argumentos para la siguiente iteracion
        copiarBloque(&bloqueIzq_pre, &bloqueIzq_post, BITS_IN_FEISTEL / 2);
        copiarBloque(&bloqueDer_pre, &bloqueDer_post, BITS_IN_FEISTEL / 2);
        copiarBloque(&bloqueIzq2_pre, &bloqueIzq2_post, BITS_IN_FEISTEL / 2);
        copiarBloque(&bloqueDer2_pre, &bloqueDer2_post, BITS_IN_FEISTEL / 2);
        copiarBloque(&clave_pre, &clave_post, BITS_IN_PC1);
  }
}
```

La segunda función es muy similar, pero en su lugar calcula los bits diferentes al cambiar la clave, realizando el proceso de DES pertinente.

```
void pruebaClaveCambiada(BloqueDES* entrada, BloqueDES* clave,
        unsigned long int* bitsCambiado, short unsigned int*** Sboxes) {

  BloqueDES bloqueIzq_pre, bloqueDer_pre, bloqueIzq_post, bloqueDer_post;
  BloqueDES bloqueTrasIP;
  BloqueDES clave_pre, clave_post, clave_ronda;
  BloqueDES clave2;
```

```
  BloqueDES clave2_pre, clave2_post, clave2_ronda;
  BloqueDES bloqueIzq2_pre, bloqueDer2_pre, bloqueIzq2_post, bloqueDer2_post;
  int i;

  //Vamos a preparar la clave cambiada
  copiarBloque(&clave2, clave, TAM_CLAVE);
  cambioBit(&clave2, TAM_CLAVE);      //Cambiamos 1 bit

  //Aplicamos la permutacion inicial a la entrada
  permInicial(&bloqueTrasIP, entrada);

  //Dividimos el bloque en dos mitades. Guardamos estas mitades
  //por duplicado para usarlos con cada clave
  getParteIzquierda(&bloqueIzq_pre, &bloqueTrasIP);
  getParteDerecha(&bloqueDer_pre, &bloqueTrasIP);
  copiarBloque(&bloqueIzq2_pre, &bloqueIzq_pre, BITS_IN_FEISTEL / 2);
  copiarBloque(&bloqueDer2_pre, &bloqueDer_pre, BITS_IN_FEISTEL / 2);

  //Aplicamos PC1
  aplicarPC1(&clave_pre, clave);
  aplicarPC1(&clave2_pre, &clave2);

  //Aplicamos las rondas DES por duplicado, una vez con la clave correcta
  //y otra con la clave con el bit cambiado
  for (i = 1; i <= NUM_ROUNDS; i++) {

        //Aplicamos LCS
        aplicarLCS(&clave_post, &clave_pre, i, 1);
        aplicarLCS(&clave2_post, &clave2_pre, i, 1);

        //Aplicamos PC2
        aplicarPC2(&clave_ronda, &clave_post);
        aplicarPC2(&clave2_ronda, &clave2_post);

        //Aplicamos F de DES
        aplicarDES(&bloqueIzq_post, &bloqueDer_post, &bloqueIzq_pre,
                    &bloqueDer_pre, &clave_ronda, i, Sboxes);
        aplicarDES(&bloqueIzq2_post, &bloqueDer2_post, &bloqueIzq2_pre, &bloqueDer2_pre,
                    &clave2_ronda, i, Sboxes);

        //Calculamos las diferencias en esta ronda en cada mitad del bloque
        bitsCambiado[i] += calculaDiferencias(&bloqueIzq_post, &bloqueIzq2_post,
        BITS_IN_FEISTEL / 2);
        bitsCambiado[i] += calculaDiferencias(&bloqueDer_post, &bloqueDer2_post,
        BITS_IN_FEISTEL / 2);

        //Prepramos los datos para otra iteracion
        copiarBloque(&bloqueIzq_pre, &bloqueIzq_post, BITS_IN_FEISTEL / 2);
        copiarBloque(&bloqueDer_pre, &bloqueDer_post, BITS_IN_FEISTEL / 2);
        copiarBloque(&bloqueIzq2_pre, &bloqueIzq2_post, BITS_IN_FEISTEL / 2);
        copiarBloque(&bloqueDer2_pre, &bloqueDer2_post, BITS_IN_FEISTEL / 2);
        copiarBloque(&clave_pre, &clave_post, BITS_IN_PC1);
        copiarBloque(&clave2_ronda, &clave2_post, BITS_IN_PC1);
  }
}
```

Resultados

Se genera el fichero de salida Avalancha.txt (mediante la instrucción: ./EfectoAvalanchaDES {-n numero de pruebas} [-o fichero de salida]), el cual nos permitirá analizar el resultado.

En nuestro caso, se ha estudiado el Efecto Avalancha a lo largo de las diferentes rondas del DES, provocado tanto por un cambio en un bit del texto de entrada como en un cambio en un bit de la clave. Por ello, es necesario comparar en cada ronda intermedia la evolución de los bloques de texto cifrado (uno con el texto y clave originales y el otro con el bit cambiado).

Para ello se generaran dos tablas con las diferencias encontradas. La primera de ellas es la de un cambio en el texto:

Ronda	Bits que cambian
0	1.00
1	3.20
2	11.90
3	24.90
4	31.60
5	30.00
6	29.30
7	33.30
8	32.50
9	29.10
10	31.60
11	32.60
12	30.80
13	31.60
14	32.90
15	31.90
16	30.20

En este caso, si nos fijamos en la tabla, el valor final es prácticamente 32 (máximo de 33,30), lo cual corresponde a la mitad del tamaño de bloque, coincidiendo con el resultado esperado. Por otro lado, cabe destacar la rápida aproximación, llegando a él en la ronda 4.

A continuación se muestra la tabla para un cambio de bit en la clave.

Ronda	Bits que cambian
0	0.00
1	1.90

2	18.80
3	31.90
4	30.10
5	27.80
6	28.20
7	30.70
8	30.80
9	31.10
10	29.50
11	29.80
12	29.30
13	29.40
14	30.40
15	30.50
16	29.90

En este caso,el valor final está en torno a 29 (un poco por encima de 28), lo cual corresponde con la mitad del tamaño de la clave sin los bits de paridad, tal y como podía esperarse.

c. Estudio de los criterios SAC y BIC

Marco teórico

- Strict Avalanche Criterion (SAC): este criterio establece que un bit de las cajas de sustitución debe cambiar con probabilidad ½ cuando se complementa con un bit de entrada, esto es:

  $P(b_j=1 \mid a_i) = P(b_j=0 \mid a_i) = ½ \ \forall i, \forall j$, con a_i bits de la caja de entrada y b_j bits de salida.

- Bit Independence Criterion (BIC): este criterio establece que un par de bits de las cajas de sustitución debe cambiar de forma independiente cuando se complementa un bit de entrada, esto es:

$P(b_j, b_k | a_i) = P(b_j | a_i) * P(b_k | a_i)$ $\forall i, \forall j, \forall k$, con a_i bits de la caja de entrada y b_j bits de salida.

Código

En este apartado, incluimos las funciones más destacadas:

La primera de ella calcula las probabilidades asociadas al SAC

```
void calcularEstadisticasSAC(SAC* datos, int nPruebas) {

  unsigned short*** Sboxes = NULL;
  unsigned long int*** cambiosDeBit = NULL;
  int i, j, k;

  srand(time(NULL));

  //Preparamos la memoria para guardar los datos de diferencias que vamos obteniendo
  cambiosDeBit = guardarMemSAC();
  for (i = 0; i < NUM_S_BOXES; i++)
        for (j = 0; j < SBOX_OUTPUT_BITS; j++)
              for (k = 0; k < SBOX_INPUT_BITS; k++)
                    cambiosDeBit[i][j][k] = 0;

  //Medimos los cambios que se producen
  Sboxes = guardarMemSboxes();
  for (i = 0; i < nPruebas; i++)
        pruebaSAC(Sboxes, cambiosDeBit);
  freeSboxes(Sboxes);

  //Calculamos las probabilidades dividiendo los casos favorables entre el numero total de casos
  for (i = 0; i < NUM_S_BOXES; i++)
        for (j = 0; j < SBOX_OUTPUT_BITS; j++)
              for (k = 0; k < SBOX_INPUT_BITS; k++)
                    datos->condProb[i][j][k] = (cambiosDeBit[i][j][k])
                                * (1. / nPruebas);

  freeMemSAC(cambiosDeBit); //Liberamos la memoria que usamos para guardar las diferencias
}
```

De manera similar, la siguiente función calcula las probabilidades necesarias en el BIC.

```
void calcularEstadisticasBIC(BIC* datos, int nPruebas) {

  unsigned short*** Sboxes = NULL;
  unsigned long int*** paresCambiados = NULL;
  int i, j, k;
  float _p1_p2, _p1p2, p1_p2, p1p2, p1, _p1, p2, _p2;

  srand(time(NULL));

  //Reservamos la memoria para ver que pares cambian
  paresCambiados = guardarMemBIC();
  for (i = 0; i < NUM_S_BOXES; i++)
        for (j = 0; j < 4 * SBOX_OUTPUT_PAIRS; j++)
              for (k = 0; k < SBOX_INPUT_BITS; k++)
                    paresCambiados[i][j][k] = 0;

  //Medimos los cambios que se producen en cada prueba
  Sboxes = guardarMemSboxes();
  for (i = 0; i < nPruebas; i++)
        pruebaBIC(Sboxes, paresCambiados);
  freeSboxes(Sboxes);

  //Calculamos las probabilidades condicionadas segun la frecuencia 'joint'
  for (i = 0; i < NUM_S_BOXES; i++)
        for (j = 0; j < SBOX_OUTPUT_PAIRS; j++)
              for (k = 0; k < SBOX_INPUT_BITS; k++) {
                    _p1_p2 = paresCambiados[i][4 * j][k] * (1. / nPruebas);
                    _p1p2 = paresCambiados[i][4 * j + 1][k] * (1. / nPruebas);
                    p1_p2 = paresCambiados[i][4 * j + 2][k] * (1. / nPruebas);
```

```
                    p1p2 = paresCambiados[i][4 * j + 3][k] * (1. / nPruebas);
                    _p1 = _p1_p2 + _p1p2;
                    p1 = p1_p2 + p1p2;
                    _p2 = _p1_p2 + p1_p2;
                    p2 = _p1p2 + p1p2;

                    datos->condProb[i][8 * j][k] = _p1_p2 * (1. / _p2);
                    datos->condProb[i][8 * j + 1][k] = _p1p2 * (1. / p2);
                    datos->condProb[i][8 * j + 2][k] = p1_p2 * (1. / _p2);
                    datos->condProb[i][8 * j + 3][k] = p1p2 * (1. / p2);
                    datos->condProb[i][8 * j + 4][k] = _p1_p2 * (1. / _p1);
                    datos->condProb[i][8 * j + 5][k] = p1_p2 * (1. / p1);
                    datos->condProb[i][8 * j + 6][k] = _p1p2 * (1. / _p1);
                    datos->condProb[i][8 * j + 7][k] = p1p2 * (1. / p1);
            }
  freeMemBIC(paresCambiados);
}
```

La tercera mide el SAC para DES en cada S-box:

```
void pruebaSAC(unsigned short*** Sboxes, unsigned long int*** cambiosDeBit) {

  BloqueDES entrada, salida;
  BloqueDES entrada2, salida2;
  int i, j, k;

  crearBloque(&entrada, SBOX_INPUT_BITS * NUM_S_BOXES);

  //En cada iteracion cambiamos los bits y ejecutamos dos SBOX, una con
  //la entrada original y otra con la entrada transformada. Despues comparamos
  //las diferencias que se han encontrado en las dos salidas
  for (k = 0; k < SBOX_INPUT_BITS; k++) {

        copiarBloque(&entrada2, &entrada, SBOX_INPUT_BITS * NUM_S_BOXES);
        cambiarDatosEntrada(&entrada2, k);

        SBoxGeneral(&salida, &entrada, Sboxes);
        SBoxGeneral(&salida2, &entrada2, Sboxes);

        for (i = 0; i < NUM_S_BOXES; i++)
              for (j = 0; j < SBOX_OUTPUT_BITS; j++)
                    if (salida2.bloque[i * SBOX_OUTPUT_BITS + j + 1]
                                != salida.bloque[i * SBOX_OUTPUT_BITS + j + 1])
                          (cambiosDeBit[i][j][k])++;
  }
}
```

La última función mide las frecuencias conjuntas para cada S-box para calcular el BIC en DES.

```
void pruebaBIC(unsigned short*** Sboxes, unsigned long int*** paresCambiados) {

  BloqueDES entrada, salida;
  BloqueDES entrada2, salida2;
  int i, j, k;
  int change1, change2;

  crearBloque(&entrada, SBOX_INPUT_BITS * NUM_S_BOXES);

  //Como en la prueba SAC volvemos a cambiar los bits de entrada
  //solo que ahora comparamos en pares
  for (k = 0; k < SBOX_INPUT_BITS; k++) {

        copiarBloque(&entrada2, &entrada, SBOX_INPUT_BITS * NUM_S_BOXES);
        cambiarDatosEntrada(&entrada2, k);

        SBoxGeneral(&salida, &entrada, Sboxes);
        SBoxGeneral(&salida2, &entrada2, Sboxes);

        for (i = 0; i < NUM_S_BOXES; i++)
              for (j = 0; j < SBOX_OUTPUT_PAIRS; j++) {
```

```
                    //Comprueba si cada par de bits de salida ha cambiado
                    //(Recordamos que se quiere la distribución conjunta de prob de cambio de cada par de bits de
salida dado
                    //un cambio en un bit de entrada, para posteriormente calcular la probabilidad condicionada de un
cambio en
                    //un bit de salida dado un cambio en uno de entrada y un cambio o no cambio en otro bit de salida)

                    if (salida2.bloque[i * SBOX_OUTPUT_BITS + OUTPUT_PAIRS[j][0] + 1]
                                != salida.bloque[i * SBOX_OUTPUT_BITS
                                            + OUTPUT_PAIRS[j][0] + 1])
                            change1 = 1;
                    else
                            change1 = 0;
                    if (salida2.bloque[i * SBOX_OUTPUT_BITS + OUTPUT_PAIRS[j][1] + 1]
                                != salida.bloque[i * SBOX_OUTPUT_BITS
                                            + OUTPUT_PAIRS[j][1] + 1])
                            change2 = 1;
                    else
                            change2 = 0;

                    //Segun que pares se haya cambiado aumentamos los datos de pares cambiados
                    if (change1 && change2)
                            (paresCambiados[i][4 * j][k])++;
                    else if (change1 && !change2)
                            (paresCambiados[i][4 * j + 1][k])++;
                    else if (!change1 && change2)
                            (paresCambiados[i][4 * j + 2][k])++;
                    else
                            (paresCambiados[i][4 * j + 3][k])++;
            }
    }
}
```

Resultados

La manera de uso será la siguiente: ./3c {-S|-B |-A} {-n numero de pruebas} [-o fichero salida], en función de si se quieren generar en un fichero los datos para el SAC (S), para el BIC (B) o para ambos (A).

Pasamos a analizar los ficheros obtenidos.

- SAC: se ha generado el fichero sac.txt. En él, se mide el SAC para cada S-box, mostrando la probabilidad de cambio en cada bit de salida cuando hay un cambio en cada bit de entrada.. En el caso de DES, las S-boxes están diseñadas para que dicha probabilidad sea superior a 1/2, sin necesidad de coincidir con dicho valor. Por tanto, analicemos las probabilidades obtenidas.

  - S1: las probabilidades obtenidas son las siguientes: 0.50, 0.75, 0.50, 0.54, 0.72, 0.74, 0.67, 0.56, 0.56, 0.75, 0.56, 0.44, 0.58, 0.73, 0.64, 0.63, 0.60, 0.57, 0.86, 0.64, 0.73, 0.60, 0.60, 0.63

  - S2: las probabilidades obtenidas son las siguientes: 0.48, 0.65, 0.82, 0.69, 0.75, 0.78, 0.91, 0.65, 0.36, 0.67, 0.76, 0.54, 0.57, 0.68, 0.58, 0.61, 0.61, 0.51, 0.50, 0.66, 0.60, 0.56, 0.53, 0.46

  - S3 las probabilidades obtenidas son las siguientes: 0.66, 0.57, 0.66, 0.77, 0.79, 0.67, 0.85, 0.72, 0.49, 0.59, 0.59, 0.68, 0.57, 0.49, 0.73, 0.60, 0.72, 0.53, 0.61, 0.91, 0.56, 0.59, 0.68, 0.71

  - S4: las probabilidades obtenidas son las siguientes: 0.62, 0.67, 0.66, 0.68, 0.60, 0.47, 0.62, 0.68, 0.67, 0.70, 0.63, 0.53, 0.68, 0.65, 0.54, 0.64, 0.60, 0.46, 0.77, 0.61, 0.69, 0.59, 0.67, 0.54

- S5: las probabilidades obtenidas son las siguientes: 0.65, 0.68, 0.67, 0.53, 0.66, 0.56, 0.71, 0.68, 0.71, 0.53, 0.76, 0.75, 0.61, 0.71, 0.62, 0.60, 0.57, 0.49, 0.64, 0.55, 0.56, 0.68, 0.54, 0.58

- S6: las probabilidades obtenidas son las siguientes: 0.70, 0.52, 0.50, 0.75, 0.57, 0.77, 0.60, 0.68, 0.77, 0.57, 0.70, 0.66, 0.74, 0.57, 0.71, 0.82, 0.74, 0.56, 0.55, 0.78, 0.48, 0.61, 0.59, 0.60

- S7: las probabilidades obtenidas son las siguientes: 0.53, 0.64, 0.69, 0.61, 0.66, 0.64, 0.61, 0.68, 0.57, 0.34, 0.90, 0.67, 0.67, 0.71, 0.76, 0.69, 0.44, 0.58, 0.85, 0.54, 0.69, 0.74, 0.78, 0.71

- S8: las probabilidades obtenidas son las siguientes: 0.75, 0.62, 0.80, 0.56, 0.58, 0.50, 0.42, 0.61, 0.49, 0.68, 0.66, 0.81, 0.64, 0.86, 0.75, 0.66, 0.36, 0.44, 0.71, 0.60, 0.67, 0.54, 0.66, 0.53

Como puede verse, el grueso de las probabilidades está comprendido entre 0.5 y 0.8, cumpliendo de este modo el principio de diseño de las S-boxes.

- BIC: de manera similar se genera el fichero bic.txt. Para el caso del BIC se ha recopilado la distribución conjunta de probabilidades de cambio de cada pareja de bits de salida dado un cambio en un bit de entrada Posteriormente, con estos datos se ha calculado la probabilidad condicionada de un cambio en un bit de salida dado un cambio en un bit de entrada y un cambio o no cambio en otro bit de salida. Debido al criterio de diseño de las S-boxes de DES, estas probabilidades son mayores de 1/2, sin necesidad de coincidir. En este caso, se ha obviado la inclusión de todas las probabilidades en la memoria, pues son muchas y así se evita extender más la memoria.

  Sin embargo, si se observa el fichero bic.txt, se puede ver que el grueso de las probabilidades para cada una de las S-boxes está por encima de ½, con alguna pequeña excepción. De este modo, se ha comprobado el correcto funcionamiento de BIC, pues coincide con el criterio de diseño indicado en el párrafo anterior.

d. Construcción de las S-boxes propias

Marco teórico y conclusiones

En este apartado se nos pide describir los criterios de diseño utilizados en el diseño de las S-boxes del algoritmo DES. Para ello, nos basamos en la información obtenida en la referencia proporcionada en el enunciado (https://pdfs.semanticscholar.org/9c48/590ecfc31f6467c8dda3937ac441de44c3a2.pdf).

Los principios de diseño son los siguientes:

- Cada S-box tiee seis bits de entrada y 4 de salida.

- Ningún bit de salida de la S-box debería estar muy cercano a una función lineal de los bits de entrada.

- Si fijamos los bits situados más a la izquierda y a la derecha de la S-box y variamos los 4 del medio, cada posible salida de 4 bits es creado exactamente una vez como el rango de los 4 bits centrales sobre las 16 posibilidades.

- Si dos entradas de la S-box difieren exactamente en un bit, la salida deberá diferir en al menos dos bits. Del mismo modo, si dos entradas de la S-box difieren exactamente en los dos bits del medio, la salida deberá diferir en al menos dos bits.

- Si dos entradas de la S-box difieren en los dos primeros bits y son idénticos en los dos últimos, entonces la salida deberá ser diferente.

- Para cualquier diferencia (que no sea cero) de 6 bits entre entradas, no más de 8 de los 32 pares de entradas puede derivar en la misma salida.

Entonces, las S-boxes propias que se han utilizado (se muestran a continuación), puede verse que siguen estos principios:

```
static const unsigned short S_BOXES[NUM_S_BOXES][ROWS_PER_SBOX][COLUMNS_PER_SBOX] = {
                                {     { 14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7 },
                                      { 0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8 },
                                      { 4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0 },
                                   { 15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13 }      },
                                                        {
                                      { 15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10 },
                                      { 3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5 },
                                      { 0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15 },
                                   { 13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9 }      },

                                {     { 10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8 },
                                      { 13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1 },
                                      { 13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7 },
                                   { 1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12 }      },

                                {     { 7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15 },
                                      { 13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9 },
                                      { 10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4 },
                                   { 3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14 }      },
                                                        {
                                      { 2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9 },
                                      { 14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6 },
                                      { 4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14 },
                                   { 11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3 }      },
                                                        {
                                      { 12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11 },
                                      { 10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8 },
                                      { 9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6 },
                                   { 4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13 }      },
                                                        {
                                      { 4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1 },
                                      { 13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6 },
                                      { 1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2 },
                                   { 6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12 }      },
                                                        {
                                      { 13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7 },
                                      { 1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2 },
                                      { 7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8 },
                                   { 2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11 }      }
                                                        };
```

Finalmente, para diseñar alguna S-box diferente bastará seguir los principios arriba indicados, como por ejemplo en el siguiente caso:

```
0, 3, 5, 6, 9, 10, 15, 12, 7, 4, 14, 13, 2, 1, 8, 11
 3, 0, 6, 5, 10, 9, 12, 15, 4, 7, 13, 14, 1, 2, 11, 8
3, 15, 0, 12, 5, 9, 10, 6, 4, 8, 11, 7, 14, 2, 1, 13
9, 5, 15, 3, 12, 0, 6, 10, 7, 11, 8, 4, 2, 14, 13, 1
```

En esta S-box se puede ver que sigue los principios necesarios para su construcción.

Ejercicio 4: Principios de diseño del AES

a. Estudio de la no linealidad de las S-boxes del AES

Marco teórico

Al igual que en el ejercicio anterior, en este apartado vamos a comprobar la no linealidad de las S-boxes, pero en este caso del AES. Su no linealidad también dificulta el criptoanálisis, al igual que en el caso del DES.

Código

Las funciones más destacadas de este apartado son (nótese que no se incluyen muchas funciones auxiliares para trabajar con polinomios que podría ser consideradas importantes, como la multiplicación o división de ellos):

La primera calcula el polinomio inverso de uno dado:

```
void calcularPolInverso(struct POLYNOMIAL* dest, struct POLYNOMIAL* s,
        struct POLYNOMIAL* mod) {
  struct POLYNOMIAL q, r, p1, p2, v1, v2;
  int d1, d2;

  iniPolinomio(&q);
  iniPolinomio(&r);
  iniPolinomio(&p1);
  iniPolinomio(&p2);
  iniPolinomio(&v1);
  iniPolinomio(&v2);

  d1 = getGrado(mod);
  d2 = getGrado(s);

  copiarPolinomios(&p1, mod);
  if (d2 >= d1)
        dividirPolinomios(&q, &p2, s, mod);
  else
        copiarPolinomios(&p2, s);

  clearPolinomio(&v1);
  clearPolinomio(&v2);
  establecerCoeficienteN(&v2, 0);

  while (1) {
        dividirPolinomios(&q, &r, &p1, &p2);
        copiarPolinomios(&p1, &p2);
        copiarPolinomios(&p2, &r);

        if (getGrado(&r) < 0)
              break;

        multiplicaPolinomios(&r, &q, &v2);
        sumaPolinomios(&r, &v1, &r);
        copiarPolinomios(&v1, &v2);
        copiarPolinomios(&v2, &r);
  }

  copiarPolinomios(dest, &v2);

  freePolinomio(&q);
  freePolinomio(&r);
  freePolinomio(&p1);
  freePolinomio(&p2);
```

```
  freePolinomio(&v1);
  freePolinomio(&v2);
}
```

La siguiente obtiene el inverso de un byte visto como un polinomio en el cuerpo de Galois:

```
uint8_t calcularInverso(uint8_t b) {
  struct POLYNOMIAL mod, p, inv;
  uint8_t ret;

  if (!b)
        return 0;

  iniPolinomio(&mod);
  iniPolinomio(&p);
  iniPolinomio(&inv);

  setPolinomio(&mod, IDENTITY, BITS_IN_BYTE);
  pasarByteAPolinomio(&p, b);

  calcularPolInverso(&inv, &p, &mod);
  ret = pasarPolinomioAByte(&inv);

  freePolinomio(&mod);
  freePolinomio(&p);
  freePolinomio(&inv);

  return ret;
}
```

La siguiente realiza la transformación afín dada una matriz y una traslación:

```
uint8_t transformacionAfin(uint8_t b, int *matriz[], int *translacion) {
  int bits[BITS_IN_BYTE];
  int bits2[BITS_IN_BYTE];
  int i, j;
  uint8_t resultado;

  //Pasamos el byte a bits
  for (i = 0; i < BITS_IN_BYTE; ++i)
        bits[i] = (b >> i) % 2;

  //Multiplicamos por la matriz modulo 2
  for (i = 0; i < BITS_IN_BYTE; ++i) {
        bits2[i] = 0;
        for (j = 0; j < BITS_IN_BYTE; ++j)
              bits2[i] += matriz[i][j] * bits[j];
        bits2[i] = bits2[i] % 2;
  }

  //Sumamos la translacion mod2
  for (i = 0; i < BITS_IN_BYTE; ++i)
        bits2[i] = (bits2[i] + translacion[i]) % 2;

  //Pasamos de bits a array
  resultado = 0;
  for (i = 0; i < BITS_IN_BYTE; ++i)
        resultado += bits2[i] << i;

  return resultado;
}
```

La última calcula las estadísticas necesarias para analizar este apartado:

```
void calcularEstadisticas(NL* resultados, int nPruebas) {

  unsigned long int coincidencias[SBOX_OUTPUT_BITS + 1];
  float aux[SBOX_OUTPUT_BITS + 1];
  struct AESBOX box;
  int i;
  float temp;
```

```
  srand(time(NULL));

  //Prepramos la estructura donde anotaremos las coincidencias
  for (i = 0; i < SBOX_OUTPUT_BITS + 1; i++)
        coincidencias[i] = 0;

  getCajaAES(&box);
  //Calculamos el porcentaje de coincidencias
  for (i = 0; i < nPruebas; i++)
        coincidencias[getDatos(&box)]++;

  //Procesa los resultados
  for (i = 0; i < SBOX_OUTPUT_BITS + 1; i++)
        aux[i] = i * (1. / SBOX_OUTPUT_BITS);

  //Calculamos el porcentaje de coincidencias
  for (i = 0; i < SBOX_OUTPUT_BITS + 1; i++)
        resultados->distribucion[i] = coincidencias[i] * (1. / nPruebas);

  //Calcula la esperanza
  resultados->esperanza = 0;
  for (i = 0; i < SBOX_OUTPUT_BITS + 1; i++)
        resultados->esperanza += (aux[i]
                    * resultados->distribucion[i]);

  //Calculamos la desviacion tipica
  resultados->desviacion = 0;
  for (i = 0; i < SBOX_OUTPUT_BITS + 1; i++) {
        temp = (aux[i] - resultados->esperanza);
        resultados->desviacion += temp * temp * resultados->distribucion[i];
  }
  resultados->desviacion = sqrt(resultados->desviacion);
}
```

Resultados

Este apartado se prueba con la siguiente instrucción: ./4a {-n numero de pruebas} [-o fichero salida], generando el fichero nl_aes.txt

La linealidad del AES también genera unos valores óptimos, como se observa en el fichero, con una esperanza de de 0,49 y una desviación típica de 0,17. Si se compara estos valores con los del ejercicio 3, en el que analizamos la linealidad del DES, se ve que son claramente inferiores, lo que refleja una menor variabilidad de los valores.

Por último, al igual que en aquel apartado, mostramos el porcentaje de coincidencias en los bits, que se resume en:

- 0 coincidencias: 1.00%
- 1 coincidencias: 3.00%
- 2 coincidencias: 12.00%
- 3 coincidencias: 20.00%
- 4 coincidencias: 27.00%
- 5 coincidencias: 25.00%
- 6 coincidencias: 10.00%
- 7 coincidencias: 2.00%
- 8 coincidencias: 0.00%

Se puede ver que sigue una distribución bastante gaussiana.

b. Generación de las S-boxes del AES

Marco teórico

Antes de hablar de las S-boxes del AES recordamos que en AES los cálculos se realizan en el cuerpo de Galois del polinomio p(x) = x 8 + x 4 + x 3 + x + 1 (cuerpo de Rijndael), el cual es un polinomio irreducible de grado 8. Esto permite que, dado un byte (esto es, un polinomio de grado 7), se pueda calcular su inversa. La forma de hacerlo será, como en la primera práctica, a través del algoritmo de Euclides, en su versión para polinomios. Para ello será necesario crear funciones de transformación entre polinomios y bytes, y funciones específicas que multiplican, suman o invierten bytes como elementos del cuerpo de arriba indicado.

Ahora ya podemos pasara a explicar la construcción de la S-box de AES.

AES se basa en dos cajas: la de encriptación y la de desencriptación, siendo la segunda una inversión de las transformaciones de la primera.

La encriptación se basa en la siguiente transformación:

S[i,j] = X*S[i,j] + C, siendo X y C una matriz y vector proporcionados en la práctica.

La desencriptación se basa en aplicar primero el inverso de la transformación afín y luego invertir como polinomio.

Código

En este apartado, mostramos las siguientes funciones:

La primera obtiene la caja de encriptación a través de una transformación afín.

```
void getCajaAES(struct AESBOX* box) {
  int i;
  for (i = 0; i < TAM_BOX; ++i)
        box->map[i] = transformacionAfin(calcularInverso(i), DIRECT_MATRIX,
                    DIRECT_TRANSLATION);
}
```

La primera obtiene la caja de desencriptación.

```
void getCajaAESInversa(struct AESBOX* box) {
  int i;
  for (i = 0; i < TAM_BOX; ++i)
        box->map[i] = calcularInverso(
                    transformacionAfin(i, INVERSE_MATRIX, INVERSE_TRANSLATION));

}
```

Esta obtiene el inverso de un byte visto como un polinomio en el cuerpo de Galois asociado al AES

```
uint8_t calcularInverso(uint8_t b) {
  struct POLYNOMIAL mod, p, inv;
  uint8_t ret;

  if (!b)
        return 0;

  iniPolinomio(&mod);
  iniPolinomio(&p);
  iniPolinomio(&inv);

  setPolinomio(&mod, IDENTITY, BITS_IN_BYTE);
  pasarByteAPolinomio(&p, b);

  calcularPolInverso(&inv, &p, &mod);
  ret = pasarPolinomioAByte(&inv);
```

```
  freePolinomio(&mod);
  freePolinomio(&p);
  freePolinomio(&inv);

  return ret;
}
```

La última, igual que en el apartado anterior, realiza la transformación afín dada una matriz y una traslación.

```
uint8_t transformacionAfin(uint8_t b, int *matriz[], int *translacion) {
  int bits[BITS_IN_BYTE];
  int bits2[BITS_IN_BYTE];
  int i, j;
  uint8_t resultado;

  //Pasamos el byte a bits
  for (i = 0; i < BITS_IN_BYTE; ++i)
        bits[i] = (b >> i) % 2;

  //Multiplicamos por la matriz modulo 2
  for (i = 0; i < BITS_IN_BYTE; ++i) {
        bits2[i] = 0;
        for (j = 0; j < BITS_IN_BYTE; ++j)
              bits2[i] += matriz[i][j] * bits[j];
        bits2[i] = bits2[i] % 2;
  }

  //Sumamos la translacion mod2
  for (i = 0; i < BITS_IN_BYTE; ++i)
        bits2[i] = (bits2[i] + translacion[i]) % 2;

  //Pasamos de bits a array
  resultado = 0;
  for (i = 0; i < BITS_IN_BYTE; ++i)
        resultado += bits2[i] << i;

  return resultado;
}
```

Resultados

Se ejecuta mediante el siguiente comando: ./4b {-C caja directa | -D caja inversa} [-o fichero salida] (caja directa y caja inversa son una explicación de lo que hacen C y D).

Por eso, en este apartado se generan los fichero directa.txt e inversa.txt

En ambos casos se ha generado la S-box correspondiente, las cuales son correctas, pues se han comparado con las cajas reales (véase las cajas del AES en Wikipedia).